5 Developing a Program

Introduction

This chapter provides an introduction to developing and executing programs for the iWarp system. Because parallel programs are highly application specific, it is impossible to provide a cookbook recipe for parallel program development. This chapter simply provides some basic examples to illustrate the fundamentals of program development and execution. These examples include

- a single-cell ‘‘Hello World” program
- a four-cell ‘‘Hello World” program
- a single-cell program using PathLib communication
- a matrix multiplication program using PathLib communication

A Single-Cell Example: “Hello World”

The following example is a simple single-cell example that illustrates the steps involved in developing and executing a program for the iWarp system. The example program is the following ‘‘Hello World” program:

```
#include <stdio.h>

main()
{
  printf(``hello world\n");
}
```

Assuming the program is stored in the file hello.c, use the following command to compile the program and store the output in the file hello.out:

```
iwcc -o hello.out  hello.c
```

Chapter 2 provides a complete description of the iwcc command.

Before you can load and execute the compiled program, you must link the program to the RTS software with the rtslink command as follows:

```
rtslink -o hello.run  hello.out
```

Chapter 3 provides a complete description of the rtslink command.

You now have an image that is ready to load and execute stored in the file hello.run. Now you must locate an array for program execution. Array names have the following form:

/dev/isib*n*

where *n* is the array number. Configuration information about each array is shown in the configuration file, /iwarp/etc/config. The following is an example of a /iwarp/etc/config file:

```
#
#  @(#)config   230.4 6/27/91 - Configuration File Change Control
#
#  This file lists each iWarp system available at this site
#
#  Each line is interpreted as follows:
#    starts with '#': comment - ignored
#    starts with '%': followed by hostname and SIB #
#    other: a keyword followed by the information for that keyword
#           keyword lines apply to the last named iWarp (last '%' line)
#           if no '%' line has occurred yet, the lines are ignored
#           lines which are REQUIRED for each machine:
#            "CELLS #"- number of array cells attached
#            "WIDTH #" and "HEIGHT #" - dimensions of the array
#            lines which are OPTIONAL for each machine:
#            "NODAK" - DAK is not present on the SIB
#            "SIBONLY" - no array is attached to the SIB (be sure to still
#                          enter CELLS, WIDTH and HEIGHT as all zero)
#
#  read via "/iwarp/bin/iwconfig" and "iwconfig()"
#

% stargazer 0
CELLS 4
WIDTH 2
HEIGHT 2
SYSHZ 16000000
LMSIZE 0x80000
ARCH 3.2
NEIGHBORMODE 0:0xe 7:0xd 64:0x0

% stargazer 1
CELLS 8
WIDTH 4
HEIGHT 2
SYSHZ 10000000
LMSIZE 0x80000
ARCH 3.2
NEIGHBORMODE
#0:0xe 1:0xd 4:0x0

#% stargazer 2
CELLS 64
WIDTH 8
HEIGHT 8
SYSHZ 10000000
LMSIZE 0x80000
ARCH 3.2
NEIGHBORMODE
```

The configuration file shows that there are three iWarp arrays. The array on /dev/isib0 is a 2 by 2, the array on /dev/isib1 is a 4 by 2, and the array on /dev/isib2 is an 8 by 8.

Before loading a program onto an array for execution, it is advisable to lock the array . If you are loading and executing the array by hand with iwmon, you must lock and unlock the array yourself. If you are using lgo to load and execute your program, lgo locks and unlocks the array for you. If you were going to load and execute your program with iwmon on /dev/isib0, you would lock the array with the following command:

```
siblock 0
```

If the lock is successful, you will get the following message:

```
SIB 0 locked (count = 1)
```

If someone else is using the array, you will get a message like the following:

```
Unable to lock SIB 0:
-rw-rw-rw-   1 hof          146 Mar 27 11:10  /usr/tmp/sib.0
---
Locked count:  2
Effective User ID:    hof
Real User ID:         iwarp!hof     ttyp9   Mar 26  14:59
Time of last lock:    Tue Mar 26  15:46:02 PST  1991
```

If you wanted to load and execute the single-cell program on cell 0 of the array, you could do so with the following lgo command:

```
lgo -r hello.run 0 -w 1 /dev/isib0
```

Chapter 3 provides a complete description of the lgo command.

When you execute the program, your output will be like the following:

```
iWarp Monitor V2.1
Mon July 1   14:20:24 PST 1991
coff V2.1
```

```
resetting the array...
booting /iwarp/lib2.1/iwmon/package.4...
iwmon> 0 mar 27 13:25:54 1991  hello world
0 Mar 27 13:25:54 1991 exit: status=0000000c
```

As can be seen from the output, lgo calls the iWarp monitor to load and run the program. You can also use the monitor to load, run, and debug your program interactively . The monitor and monitor commands are described in Chapter 3.

The output shows that any output that comes to the terminal is tagged with the id of the cell and the date as seen in the line:

```
iwmon> 0 mar 27 13:25:54 1991  hello world
```

When the program exits, the exit status is printed. Our “hello world” program did not call exit, so our exit status is not meaningful.

After running the program, you should unlock the array for use by others. To unlock /dev/isib0, use the sibunlock command as follows:

```
sibunlock 0
```

“Hello World” on Four Cells

The ‘‘hello world” program for four cells would be the following:

```
#include <stdio.h>

int _cellid;

main()
{
   fprintf(stdout, "hello world from cell %d\n", _cellid);
   exit(0);
}
```

In this program we’ve added a call to exit and a variable called _cellid. This variable receives the logical cell id of a cell from the runtime system. To compile, link, and run the program on 4 cells on the array on /dev/isib0, enter the following commands (% represents the Sun prompt):

```
% iwcc -o hello.out hello.c

% rtslink -o hello.run hello.out

% lgo -r hello.run 0 1 2 3 -w 4 /dev/isib0
```

```
iWarp Monitor V2.1
Mon July 1  14:20:24 PST 1991
coff V2.1
```

```
resetting the array...
booting /iwarp/lib2.1/iwmon/package.4...
iwmon> 0 mar 27 13:25:54 1991  hello world from cell 0
2 mar 27 13:25:54 1991  hello world from cell 2
3 mar 27 13:25:54 1991  hello world from cell 3
1 mar 27 13:25:54 1991  hello world from cell 1
iwmon> %
```

A Single-Cell Program Using PathLib Communication

The examples shown so far have illustrated the mechanics of executing programs on an iWarp array, but they have not discussed the most challenging aspect of parallel programming on the iWarp system; program communication between cells. Chapter Four provided an introduction to the PathLib library of communication functions. This library provides functions to implement program-based communication in programs developed for the iWarp system. Appendix B also provides descriptions of the PathLib functions.

This section provides an example of a single-cell program that uses PathLib functions to communicate with itself. In this example we are assuming a two by two array , configured as shown in Figure 5-1.

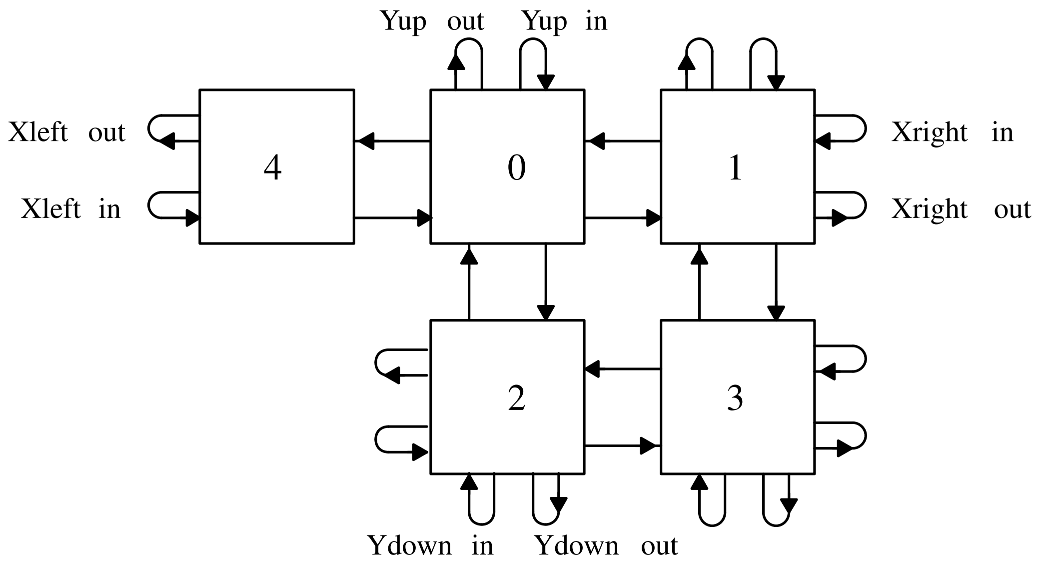

Figure 5-1: Two by Two Array Configuration

Note that cell 4 in the configuration is the interface cell, and cells 0 through 3 are the array cells. In general, with an N-cell system, the array cells are numbered 0 through N–1, and the interface cell is numbered N.

For this example, the program is loaded and run on cell 0. Cell 0 sends out a channel to the right, and the channel returns to cell 0 on the left after passing through cell 1 and cell 4. The following is the example program.

```
#include <stdio.h>
#include <iwarp_cc.h>
#include <gates.h>
#include <regnums.h>
#include <asm/gen_asm.h>
#include <asm/pw_asm.h>
#include <pathlib/pl.h>

int _cellid;
```

```
main()
{
  int o_chan;                    /*  logical channel descriptor */
  int nr_corners;                /*  number of corners to turn to
                        reach destination */
  int dest_id;                   /*  name of the destination cell */

  int i_chan;                    /*  logical channel descriptor */
  int header;                    /*  storage for input header */
  int i;
  int tmp;

  /* Initialize the PL module */
  PL_INIT(_cellid, 0xfffc, 0xff);

  /* Set up PCTs for neighbors */
  pl_rpe_configure(0x0030, 0x00c0, 0x0300, 0x0c00, 0xf000);

  /* Assume the program is to run on cell 0 only */
  if (_cellid != 0) {
    exit(0);
  }

  /* Create a logical channel to this cell in the Xright direction */
  dest_id = _cellid;
  nr_corners = 1;

  o_chan = pl_create_connection(GATE0_OUT, PL_DIR_XR, &dest_id,
                                    nr_corners);
  (void) pl_produce_message_header(GATE0_OUT,dest_id);

  /* Accept the logical channel just created. */
  i_chan = pl_accept_connection(GATE1_IN, &header);
  (void) pl_consume_message_header(GATE1_IN,&header);

  /* Send and receive 1000 words over the channel */
    for (i=0; i<1000; i++) {
      _sendi(GATE0_OUT, i);
      tmp = _receivei(GATE1_IN);
      if (tmp != i) }
        fprintf(stderr, "error: sent %d, recd %d\n", i, tmp);
        exit(1);
      }
    }

  /* Terminate the output channel */
  (void) pl_produce_message_trailer(GATE0_OUT);
  (void) pl_destroy_connection(o_chan,GATE0_OUT);

  /* Terminate the input channel */
  (void) pl_consume_message_trailer(GATE1_IN);
  (void) pl_conclude_connection(i_chan,GATE1_IN);

  exit(0);
}
```

A section by section discussion of the example is helpful to understanding how PathLib communication works.

```
/* Initialize the PL module */
PL_INIT(_cellid, 0xfffc, 0xff);
```

PL_INIT initializes the PathLib module. PL_INIT told PathLib that it could access 18 of the 20 PCTs, with the argument 0xfffc defining which ones. The two PCTs allocated to iwmon, PCT0 and PCT1 are not available for use by PathLib. PL_INIT also tells PathLib which spoolgates are to be used. There is a maximum of eight, and in this example the argument 0xff is a bit mask indicating that all of the spoolgates are to be used.

```
/* Set up PCTs for neighbors */
pl_rpe_configure(0x0030, 0x00c0, 0x0300, 0x0c00, 0xf000);
```

pl_rpe_configure estalishes the allocation of logical channel resources for neighbor cells. The first four arguments to the function represent RPEs for the four directions, Xleft, Xright, Yup, and Ydown. The last argument defines which RPE is used for output connections. The arguments to pl_rpe_configure are bit masks. representing which RPEs to use. There are a total of 20 RPEs, numbered 0 through 19. Examining the arguments we see that for this example:

- Xleft uses RPEs 4 and 5
- Xright uses RPEs 6 and 7
- Yup uses RPEs 8 and 9
- Ydown uses RPEs 10 and 11
- RPEs 12-15 are used for output connections.

Note that RPEs 0 and 1 are reserved for the monitor. RPEs 2, 3, and 16-19 are not used in this example.

```
/* Create a logical channel to this cell in the Xright direction */
dest_id = _cellid;
nr_corners = 1;

o_chan = pl_create_connection(GATE0_OUT, PL_DIR_XR, &dest_id,
                              nr_corners);
(void) pl_produce_message_header(GATE0_OUT,dest_id);
```

pl_create_connection creates a connection bound to a stream gate routed in an initial direction with routing information. Looking at each of the arguments:

GATE0_OUT	A stream gate. There are two gates available, GATE0 and GATE1. Each gate has an input and an output side.When you write (send) to the output, the data is written out to the logical channel to which the output side of the gate is bound. When you read from the input side, data is read (receive) from the logical channel to which the input side of the gate is bound.
PL_DIR_XR	Specifies the initial direction of the connection, in this case, Xright. This argument determines the direction of the first routing control word.
dest_id	an array of routing control words.
nr_corners	How many routing control words are in the array.

pl_produce_message_header sends a message header via the outbound connection. GATE0_OUT is the streamgate bound to the output PCT entry that is the origin of the connection. dest_id specifies the cell address.

We have created a connection to ourself and have sent a message, now we must accept the message.

```
/* Accept the logical channel just created. */
i_chan = pl_accept_connection(GATE1_IN, &header);
(void) pl_consume_message_header(GATE1_IN,&header);
```

pl_accept_connection consumes the header for the connection and returns it in i_chan. pl_consume_message_header consumes the message header. We are now ready to send and receive data.

```
/* Send and receive 1000 words over the channel */
  for (i=0; i<1000; i++) {
    _sendi(GATE0_OUT, i);
    tmp = _receivei(GATE1_IN);
    if (tmp != i) }
      fprintf(stderr, "error: sent %d, recd %d\n", i, tmp);
      exit(1);
    }
  }
```

In the for loop, we send data from GATE0_OUT , and receive from GATE1_IN. When the for loop is complete, we must clean up the connections.

```
/* Terminate the output channel */
(void) pl_produce_message_trailer(GATE0_OUT);
(void) pl_destroy_connection(o_chan,GATE0_OUT);
```

pl_produce_message_trailer closes the message, and pl_destroy_connection closes the output connection.

```
/* Terminate the input channel */
(void) pl_consume_message_trailer(GATE1_IN);
(void) pl_conclude_connection(i_chan,GATE1_IN);
```

pl_consume_message_trailer consumes the message trailer and pl_conclude_connection closes the input connection.

We now compile the program:

```
iwcc -o example.out example.c
```

Since we are going to run the program on the interface cell as well as the array cells, we have to do two seperate links to the runtime system:

```
rtslink -sib -o example.sib.run example.out
```

```
rtslink -o example.run example.out
```

Next, we generate a run script with lgo:

```
lgo -n -o lgo.script -R example.sib.run 4 -r example.run 0..3 /dev/isib0
```

This command generates the following file, lgo.script:

```
load 4 example.sib.run
run 4 0x20000
load 0 example.run
run 0 0x12000
load 1 example.run
run 1 0x12000
load 2 example.run
run 2 0x12000
load 3 example.run
run 3 0x12000
exit
```

In this example, it is critical that all cells have called PL_INIT and pl_rpe_configure before cell 0 starts processing. To do this, we can edit lgo.script to ensure that cells 4, 1, 2, and 3 have started and exited before cell 0 starts. The following is the edited lgo.script:

```
load 4 example.sib.run
run 4 0x20000
load 1 example.run
run 1 0x12000
load 2 example.run
run 2 0x12000
load 3 example.run
run 3 0x12000
wait 4
load 0 example.run
run 0 0x12000
wait 1
exit
```

We can now load and execute the program by feeding lgo.script back into lgo as follows:

```
lgo –f lgo.script /dev/isib0
```

This example illustrates the importance of inter-cell synchronization. In this case it was easy to synchronize the cells, but that is not always the case. There is no general mechanism to handle inter–cell synchronization, and it must be thought out on an application by application basis.

Matrix Multiplication Using PathLib Communication

The following example solves a matrix multiplication on a two by two iWarp array. The matrices are four by four. Figure 5-2 shows the contents of matrix A and B, and the desired contents of the result matrix, matrix C.

$$
\begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ \hline \end{array}
\;\mathrm{X}\;
\begin{array}{|c|c|c|c|}\hline 1 & 1 & 1 & 1 \\ \hline 1 & 1 & 1 & 1 \\ \hline 1 & 1 & 1 & 1 \\ \hline 1 & 1 & 1 & 1 \\ \hline \end{array}
\;=\;
\begin{array}{|c|c|c|c|}\hline 10 & 10 & 10 & 10 \\ \hline 26 & 26 & 26 & 26 \\ \hline 42 & 42 & 42 & 42 \\ \hline 58 & 58 & 58 & 58 \\ \hline \end{array}
$$

Matrix A　　　　Matrix B　　　　Matrix C

Figure 5-2: Contents of Matrices for Matrix Multiplication

Our strategy is to distribute one row of matrix A to each cell in the array. Each cell can then compute a row of the result matrix. When each cell finishes computing, the result rows will be collected to compile the result. Figure 5-3 shows our four-cell configuration.

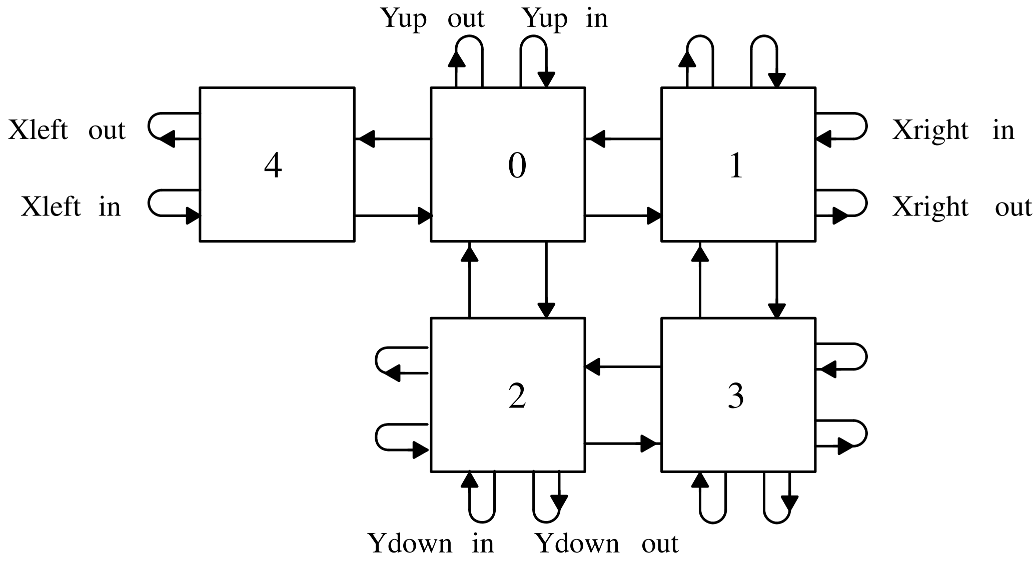

Figure 5-3: Two by Two Array Configuration

We will compile the result matrix on cell 0, using the following paths to return the results from the other cells:

- cell 1: Xleft to cell 0
- cell 2: Yup to cell 1, Xleft to cell 0
- cell 2: Xright to cell 3, Yup to cell 1, Xleft to cell 0

The following is the program.

```
#include <stdio.h>
#include <iwarp_cc.h>
#include <gates.h>
#include <regnums.h>
#include <asm/gen_asm.h>
#include <asm/pw_asm.h>
#include <pathlib/pl.h>

int mata[4][4] = {
      1,  2,  3,  4,
      5,  6,  7,  8,
      9, 10, 11, 12
     13, 14, 15, 16,
};

int matb[4][4] = {
      1, 1, 1, 1,
      1, 1, 1, 1,
      1, 1, 1, 1
      1, 1, 1, 1,
};

int matc[4][4] = {
      0, 0, 0, 0,
      0, 0, 0, 0,
      0, 0, 0, 0
      0, 0, 0, 0,
};

int    _cellid;

void   do_row();
void   collect_results();
void   send_results();

main()
{
  int i;
  int j;

  /* Initialize the PL module */
  PL_INIT(_cellid, 0xfffc, 0xff);

  /* Set up PCTs for neighbors */
  pl_rpe_configure(0x0030, 0x00c0, 0x0300, 0x0c00, 0xf000);

  /* Distribute Matrix A */
  do_row(_cellid);

  if (_cellid == 0) {
    collect_results();
    fprintf(stdout, "\nresult of matrix A * matrix B\n\n"
    for (i=0; i<4; i++) {
      for (j=0; j<4; j++) {

        fprintf(stdout, "%2d ", matc[i][j]);
      }
```

```
      fprintf(stdout, "\n");
    }
    fprintf(stdout, "\n");
  }
  else {
    send_results();
  }

  exit(0);


void
do_row(row)
  int row;
{

  int col;
  int i;
  int sum;
  for (col=0; col<4; col++) {
    sum = 0;
    for (i=0; i<4; i++) {
      sum += mata[row][i] * matb[i][col];
    }
    matc[row][col] = sum;
  }
}



void
collect_results()
{

  int i_chan;
  int header;
  int fromid;
  int i;
  int j;

  for (i=1; i<4; i++) {


    /* Accept the logical channel */
    i_chan = pl_accept_connection(GATE1_IN, &header);
    (void) pl_consume_message_header(GATE1_IN, &header);

    /* First data word in cellid of sender */
    fromid = _receivei(GATE1_IN);

    for (j=0; j<4; j++) {
      matc[fromid][j] = _receivei(GATE1_IN);
    }
```

```
    /* Terminate the input channel */
    (void)pl_consume_message_trailer(GATE1_IN);
    (void)pl_conclude_connection(i_chan,GATE1_IN);
  }
}


  /* cell 1: xl to cell 0
     cell 3: yu to cell 1, xl to cell 0
     cell 2: xr to cell 3, yu to cell 1, xl to cell 0  */

void
send_results()
{
  int o_chan;
  int headers[4];
  int header_count;
  int i;

  switch (_cellid) {

  case 1:
     /*
     * asm_build_header
     *      dest:   0
     *      turn:   0=no, 1=yes
     *      pw:     0=X, 1=Y
     *      dir:    0=left, 1=right
     *      pool:   0, 1
     */
     headers[0] = asm_build_header(0, 0, 0, 0, 0);
     header_count = 1;
     o_chan = pl_create_connection(GATE0_OUT, PL_DIR_XL,
                                   headers, header_count);
     break;

  case 3:
     headers[0] = asm_build_header(1, 1, 0, 0, 0);
     headers[1] = asm_build_header(0, 0, 0, 0, 0);
     header_count = 2;
     o_chan = pl_create_connection(GATE0_OUT, PL_DIR_YU,
                                   headers, header_count);
     break;

  case 2:
     headers[0] = asm_build_header(3, 1, 1, 1, 0);
     headers[1] = asm_build_header(1, 1, 0, 0, 0);
     headers[2] = asm_build_header(0, 0, 0, 0, 0);
     o_chan = pl_create_connection(GATE0_OUT, PL_DIR_XR,
                                   headers, header_count);
     break;
  }
```

```
  (void)pl_produce_message_header(GATE0_OUT,0);

  /* Let receiver know what cell data is coming from */

  _sendi(GATE0_OUT, _cellid);

  /* Send result row */

  for (i=0; i<4; i++) {
    _sendi(GATE0_OUT,matc[_cellid][i]);
  }

  /* Terminate the output channel */

  (void)pl_produce_message_trailer(GATE0_OUT);
  (void)pl_destroy_connection(o_chan,GATE0_OUT);
}
```

As with our previous example, we compile, link to the RTS, and generate a command script for program execution.

```
iwcc -o matmult.out matmult.c

rtslink -o matmult.run matmult.out

lgo -n -o matmult.script -R matmult.sib.run 4 -r matmult.run 0..3 \
                              /dev/isib1
```

The output of the run script generated by lgo is the following:

```
load 4 matmult.sib.run
run 4 0x20000
load 0 matmult.run
run 0 0x12000
load 1 matmult.run
run 1 0x12000
load 2 matmult.run
run 2 0x12000
load 3 matmult.run
run 3 0x12000
exit
```

We must edit the script as follows to provide the correct starting synchronization for the cells:

```
load 4 matmult.sib.run
run 4 0x20000
load 0 matmult.run
run 0 0x12000
load 1 matmult.run
run 1 0x12000
load 3 matmult.run
run 3 0x12000
load 2 matmult.run
run 2 0x12000
wait 5
exit
```

Now we can load and execute the program.

```
%lgo -f matmult.script /dev/isib1
```

iWarp Monitor V2.1 Mon July 1 14:20:24 PST 1991 coff V2.1

```
resetting the array...
booting /iwarp/lib2.1/iwmon/package.4...
iwmon> 0 mar 27 13:25:54 1991
result of matrix A * matrix B

  0 Mar 27 13:25:54 1991  10 10 10 10
  0 Mar 27 13:25:54 1991  26 26 26 26
  0 Mar 27 13:25:54 1991  42 42 42 42
  0 Mar 27 13:25:54 1991  58 58 58 58
  0 Mar 27 13:25:54 1991
iwmon> %
```